Overview of the Global Arrays Parallel Software Development Toolkit

Bruce Palmer

Jarek Nieplocha, Manoj Kumar Krishnan, Vinod Tipparaju

Pacific Northwest National Laboratory

Overview

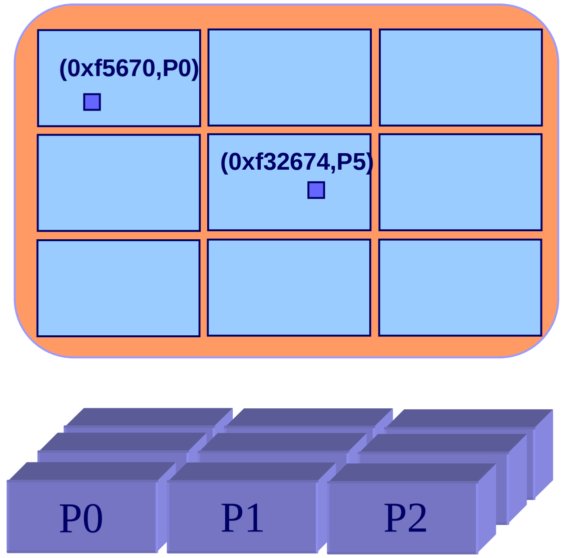
- Background
- Programming Model
- Core Capabilities
- New Functionality
- Applications
- Summary

Distributed Data vs Shared Memory

Distributed Data:

Data is explicitly associated with each processor, accessing data requires specifying the location of the data on the processor and the processor itself.

Data locality is explicit but data access is complicated. Distributed computing is typically implemented with message passing (e.g. MPI)

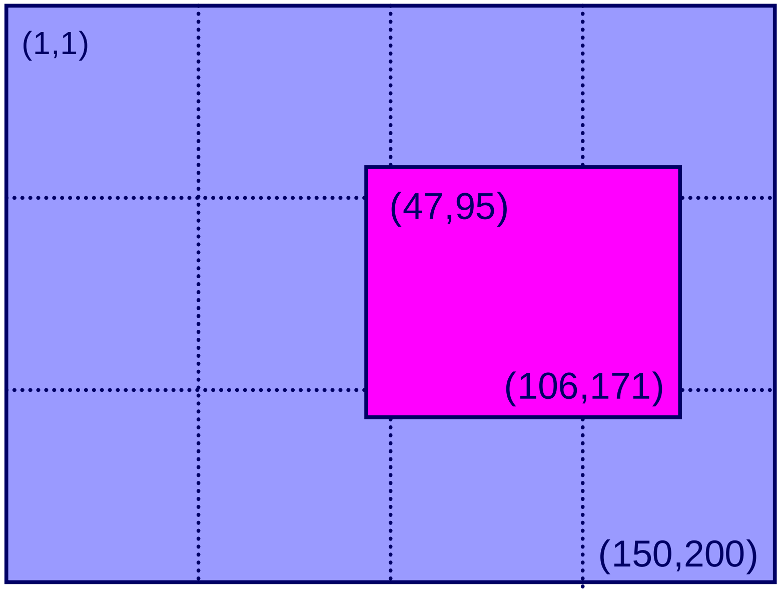


Distributed Data vs Shared Memory (Cont).

Shared Memory:

Data is an a globally accessible address space, any processor can access data by specifying its location using a global index

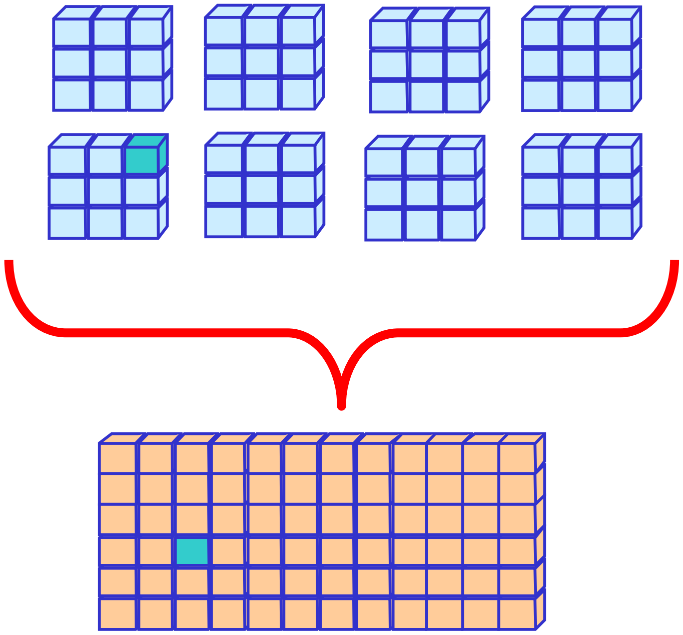
Data is mapped out in a natural manner (usually corresponding to the original problem) and access is easy. Information on data locality is obscured and leads to loss of performance.



Global Arrays

Distributed dense arrays that can be accessed through a shared memory-like style

Physically distributed data



Global Address Space

single, shared data structure/ global indexing

e.g., access A(4,3) rather than buf(7) on task 2

Global Arrays (cont.)

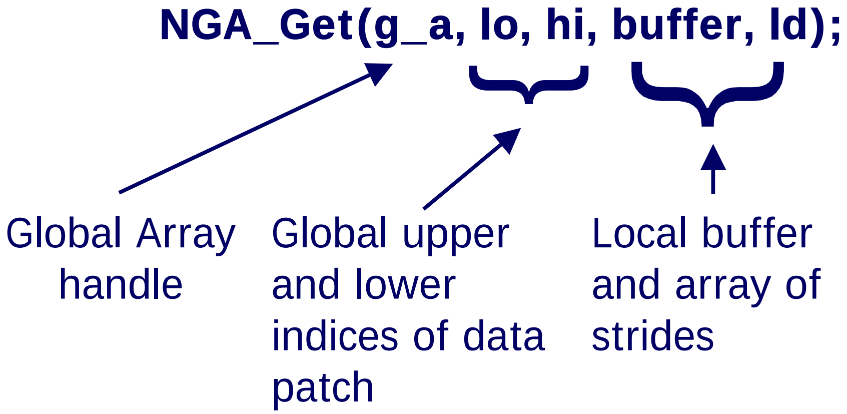
- Shared memory model in context of distributed dense arrays
- Much simpler than message-passing for many applications
- Complete environment for parallel code development
- Compatible with MPI
- Data locality control similar to distributed memory/message passing model
- Extensible
- Scalable

Remote Data Access in GA

Message Passing:

```
identify size and location of data blocks

loop over processors:
    if (me = P_N) then
        pack data in local message buffer
        send block of data to message buffer on P0
    else if (me = P0) then
        receive block of data from P_N in message buffer
        unpack data from message buffer to local buffer
    endif
end loop

copy local data on P0 to local buffer
```

Global Arrays:

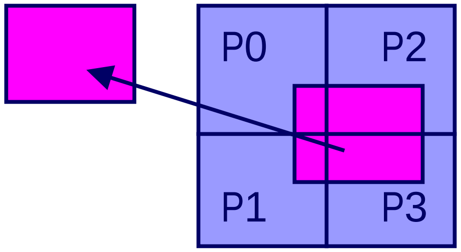


NGA_Get(g_a, lo, hi, buffer, ld);

Global Array handle

Global upper and lower indices of data patch

Local buffer and array of strides



Data Locality

What data does a processor own?

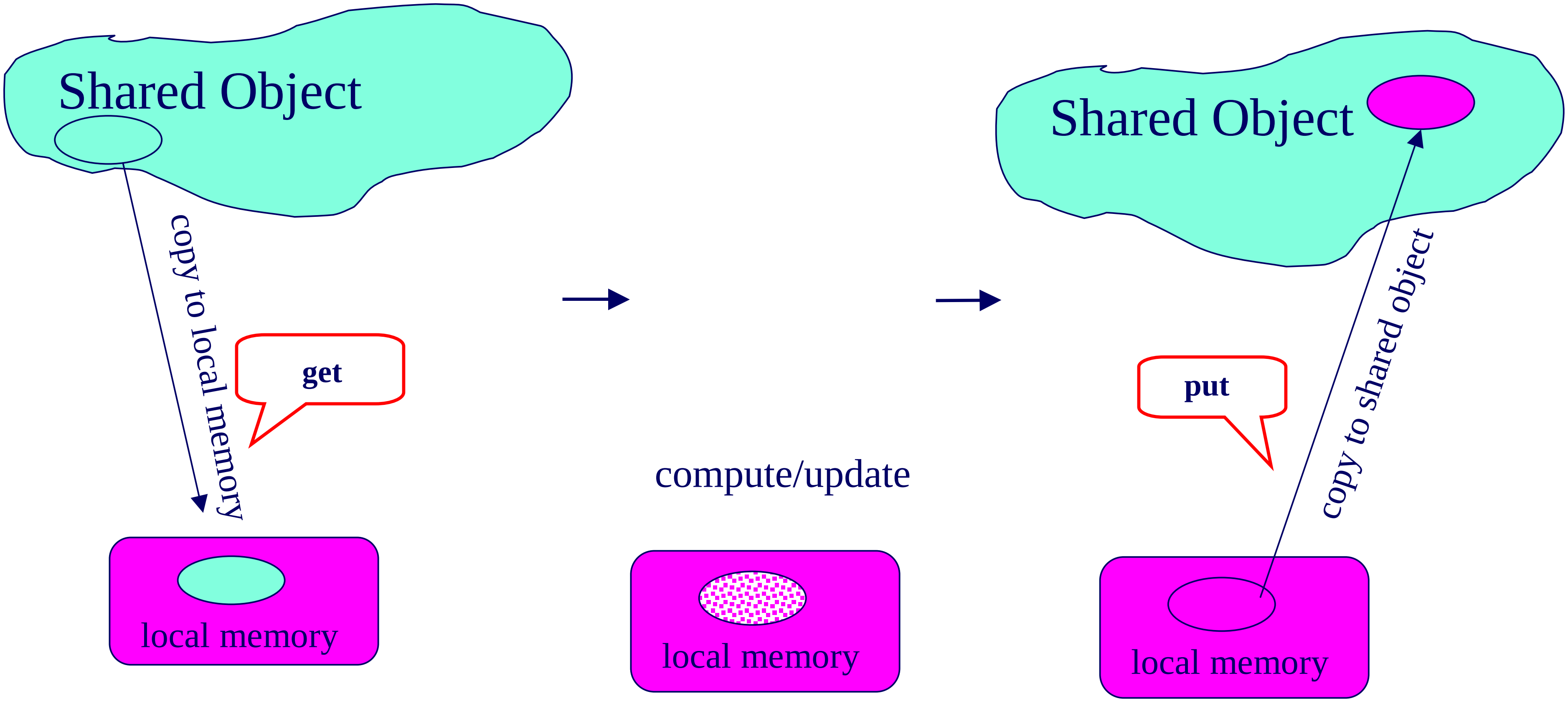
```
NGA_Distribution(g_a, iproc, lo, hi);
```

Where is the data?

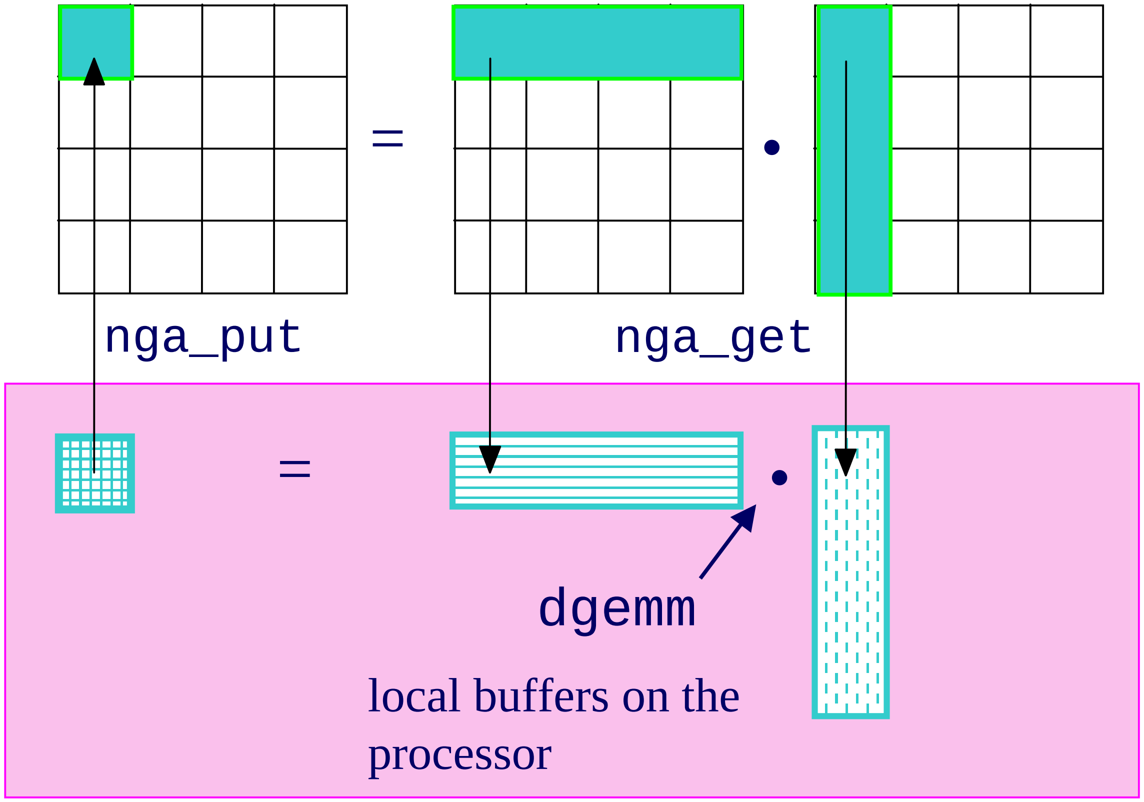
```
NGA_Access(g_a, lo, hi, ptr, ld)
```

Use this information to organize calculation so that maximum use is made of locally held data

Global Array Model of Computations



Example: Matrix Multiply



global arrays representing matrices

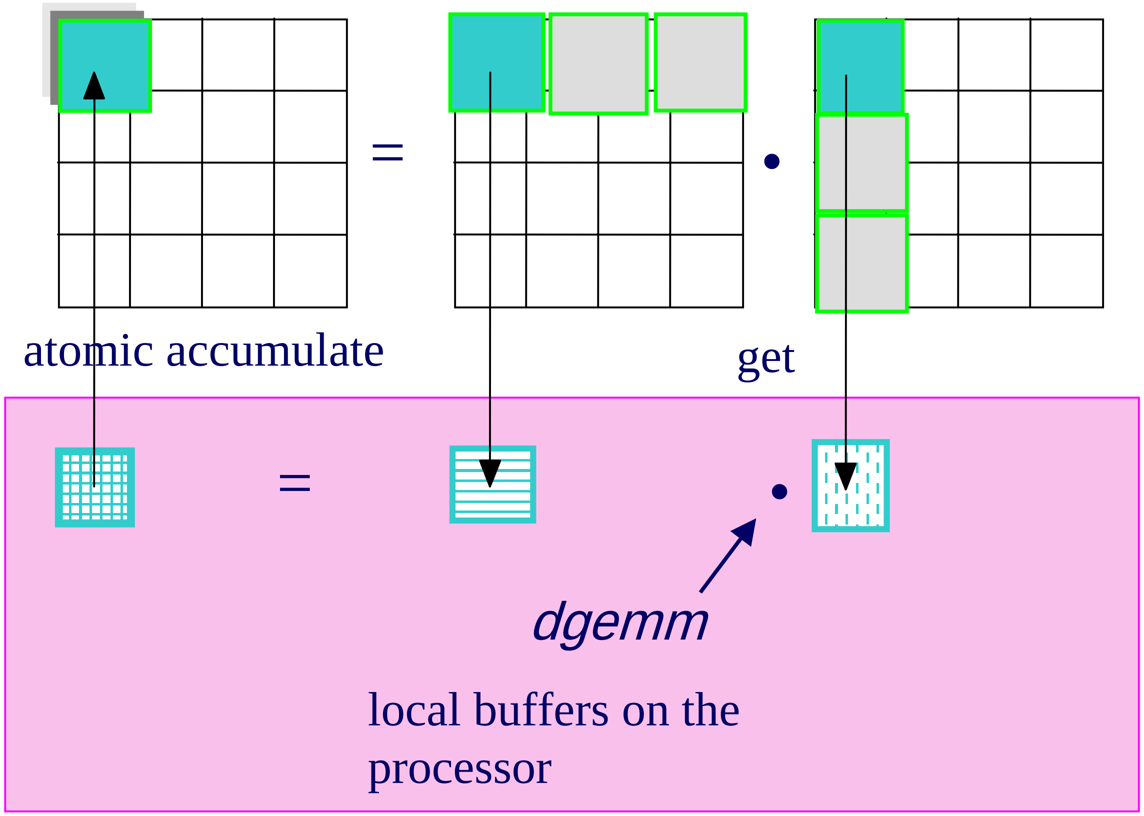
nga_put

nga_get

dgemm

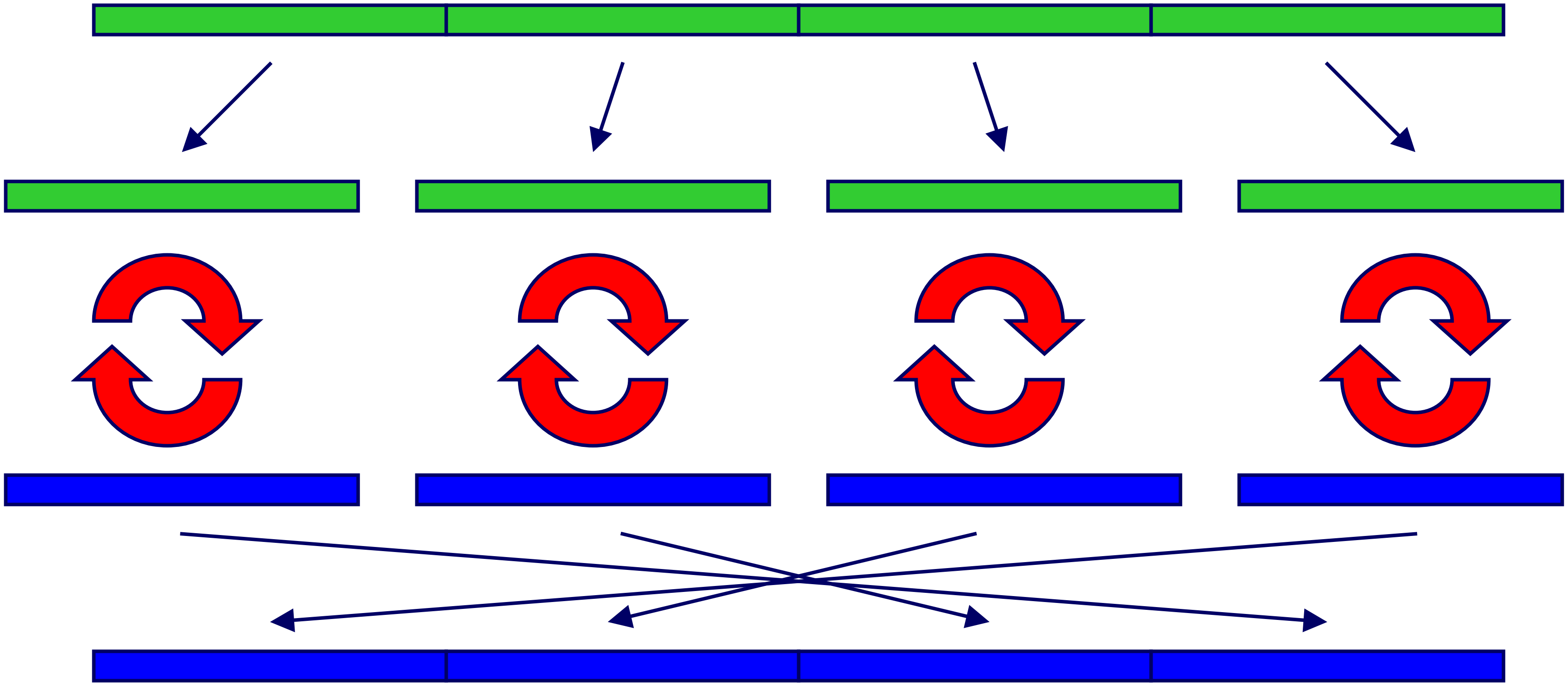
local buffers on the processor

Matrix Multiply
(a better version)



more scalable!
(less memory,
higher parallelism)

Example: 1-D Transpose



Example: 1-D Transpose (cont.)

```
#define   NDIM         1
#define   TOTALELEMS   197
#define   MAXPROC   128
      program main
      implicit none
#include "mafdecls.fh"
#include "global.fh"

      integer dims(3), chunk(3), nprocs, me, i, lo(3), hi(3), lo1(3)
      integer hi1(3), lo2(3), hi2(3), ld(3), nelem
      integer g_a, g_b, a(MAXPROC*TOTALELEMS), b(MAXPROC*TOTALELEMS)
      integer heap, stack, ichk, ierr
      logical status
      heap = 300000
      stack = 300000
```

Example: 1-D Transpose (cont.)

```
c     initialize communication library
      call mpi_init(ierr)
c     initialize ga library
      call ga_initialize()
      me = ga_nodeid()
      nprocs = ga_nnodes()
      dims(1)   = nprocs*TOTALELEMS + nprocs/2  ! Unequal data distribution
      ld(1) = MAXPROC*TOTALELEMS
      chunk(1) = TOTALELEMS  ! Minimum amount of data on each processor
      status = ma_init(MT_F_DBL, stack/nprocs, heap/nprocs)

c     create a global array
      status = nga_create(MT_F_INT, NDIM, dims, "array A", chunk, g_a)
      status = ga_duplicate(g_a, g_b, "array B")

c     initialize data in GA
      do i=1, dims(1)
         a(i) = i
      end do
      lo1(1) = 1
      hi1(1) = dims(1)
      if (me.eq.0) call nga_put(g_a,lo1,hi1,a,ld)
      call ga_sync() ! Make sure data is distributed before continuing
```

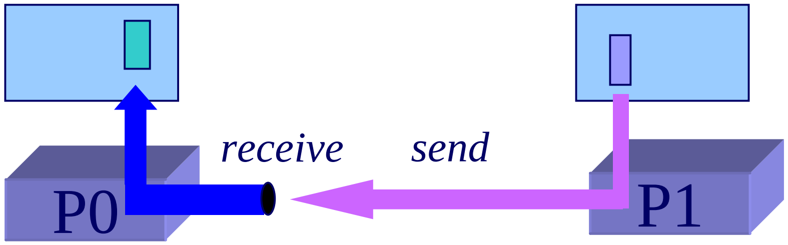
Example: 1-D Transpose (cont.)

```
c     invert data locally
      call nga_distribution(g_a, me, lo, hi)
      call nga_get(g_a, lo, hi, a, ld) ! Use locality
      nelem = hi(1)-lo(1)+1
      do i = 1, nelem
        b(i) = a(nelem - i + 1)
      end do

c     invert data globally
      lo2(1) = dims(1) - hi(1) + 1
      hi2(1) = dims(1) - lo(1) + 1
      call nga_put(g_b,lo2,hi2,b,ld)
      call ga_sync() ! Make sure inversion is complete
```

Example: 1-D Transpose (cont.)

```
c     check inversion
      call nga_get(g_a,lo1,hi1,a,ld)
      call nga_get(g_b,lo1,hi1,b,ld)
      ichk = 0
      do i= 1, dims(1)
        if (a(i).ne.b(dims(1)-i+1).and.me.eq.0) then
          write(6,*) "Mismatch at ",i
          ichk = ichk + 1
        endif
      end do
      if (ichk.eq.0.and.me.eq.0) write(6,*) "Transpose OK"

      status = ga_destroy(g_a) ! Deallocate memory for arrays
      status = ga_destroy(g_b)
      call ga_terminate()
      call mpi_finalize(ierr)
      stop
      end
```

One-sided Communication

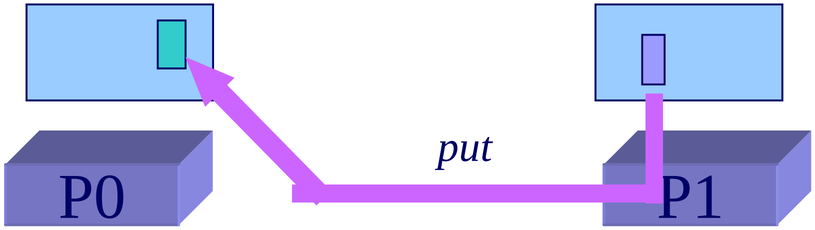


message passing

MPI

Message Passing:

Message requires cooperation on both sides. The processor sending the message (P1) and the processor receiving the message (P0) must both participate.



one-sided communication

SHMEM, ARMCI, MPI-2-1S

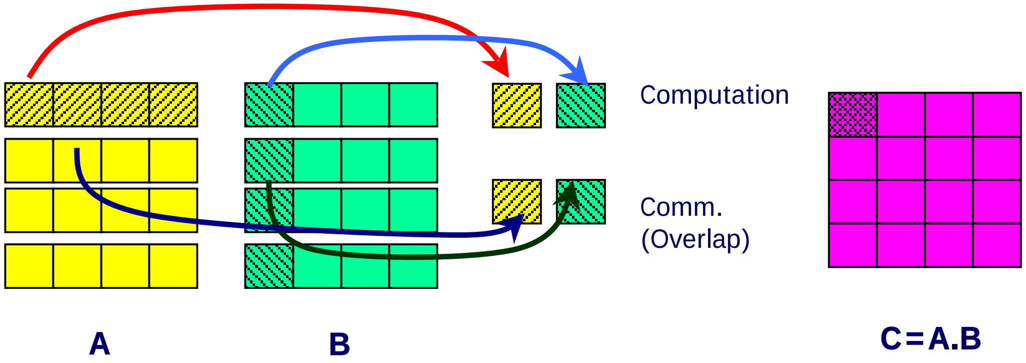
One-sided Communication:

Once message is initiated on sending processor (P1) the sending processor can continue computation. Receiving processor (P0) is not involved.

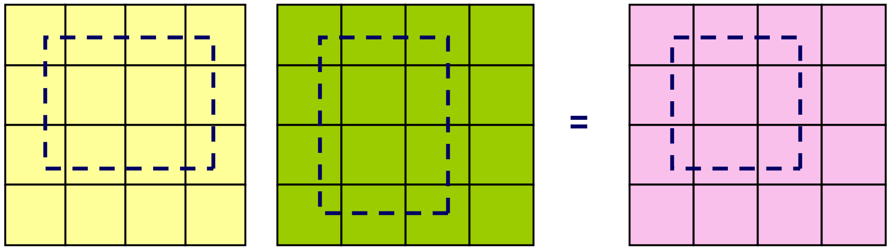
Non-Blocking Communication

- New functionality in GA version 3.3
- Allows overlapping of data transfers and computations
  - Technique for latency hiding
- Nonblocking operations initiate a communication call and then return control to the application immediately
- operation completed locally by making a call to the *wait* routine

SUMMA Matrix Multiplication



```
Issue NB Get A and B blocks
do (until last chunk)
   issue NB Get to the next blocks
   wait for previous issued call
   compute A*B (sequential dgemm)
   NB atomic accumulate into “C”
      matrix
done
```



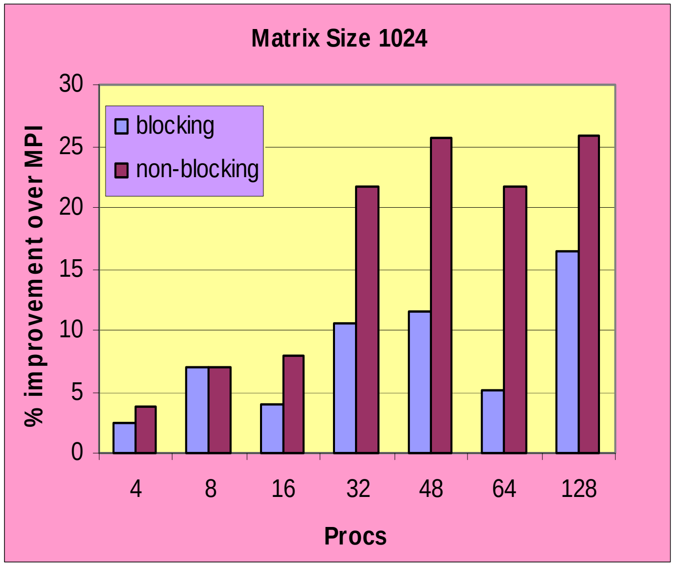
patch matrix multiplication

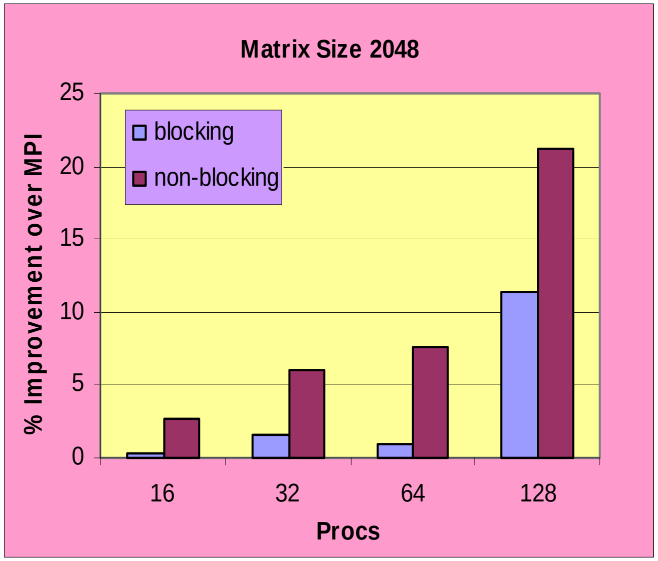
Advantages:

- Minimum memory
- Highly parallel
- Overlaps computation and communication
    - latency hiding
- exploits data locality
- patch matrix multiplication (easy to use)
- dynamic load balancing

SUMMA Matrix Multiplication: Improvement over MPI

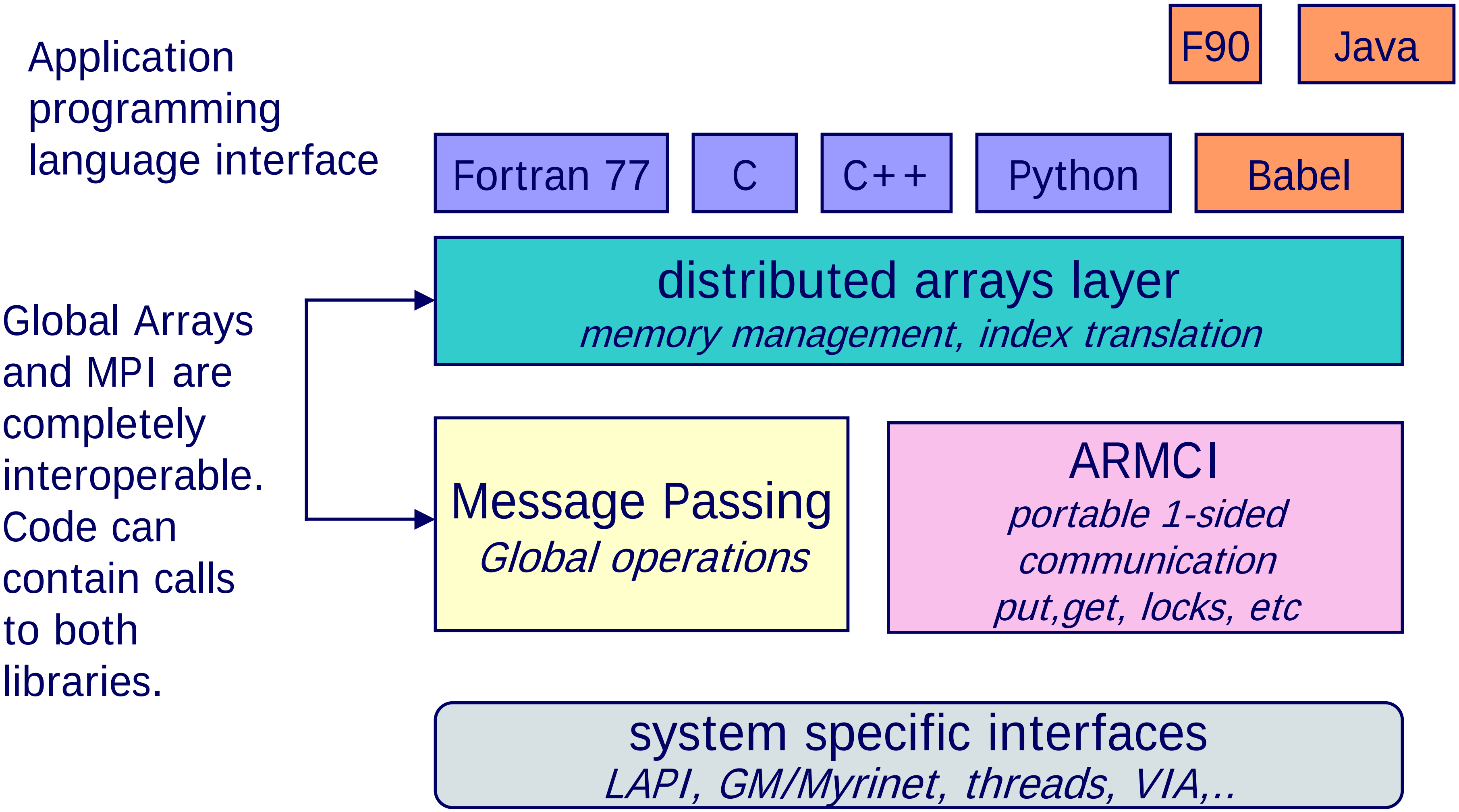
Non-Blocking Communication Performance





***2.4Ghz P4 Linux cluster, Myrinet-GM interconnect (at SUNY, Buffalo)**

Structure of GA



Application programming language interface

F90

Java

Fortran 77

C

C++

Python

Babel

distributed arrays layer

memory management, index translation

Global Arrays and MPI are completely interoperable. Code can contain calls to both libraries.

Message Passing

Global operations

ARMCI

portable 1-sided communication

put,get, locks, etc

system specific interfaces

LAPI, GM/Myrinet, threads, VIA,..

Core Capabilities

- Distributed array library
  - dense arrays 1-7 dimensions
  - four data types: *integer*, *real, double precision*, *double complex*
  - global rather than per-task view of data structures
  - user control over data distribution: regular and irregular
- Collective and shared-memory style operations
  - ga_sync, ga_scale, etc
  - ga_put, ga_get, ga_acc
  - nonblocking ga_put, ga_get, ga_acc
- Interfaces to third party parallel numerical libraries
  - PeIGS, Scalapack, SUMMA, Tao
    - ☒ example: to solve a linear system using LU factorization

```
call ga_lu_solve(g_a, g_b)
```

instead of

```
call pdgetrf(n,m, locA, p, q, dA, ind, info)
call pdgetrs(trans, n, mb, locA, p, q, dA,dB,info)
```

Interoperability and Interfaces

- Language interfaces to Fortran, C, C++, Python
- Interoperability with MPI and MPI libararies
  - e.g., PETSC, CUMULVS
- Explicit interfaces to other systems that expand functionality of GA
  - ScaLAPACK-scalable linear algebra software
  - Peigs-parallel eigensolvers
  - TAO-advanced optimization package

Global Array Processor Groups

Many parallel applications require the execution of a large number of independent tasks. Examples include

- Numerical evaluation of gradients

- Monte Carlo sampling over initial conditions or uncertain parameter sets

- Free energy perturbation calculations (chemistry)

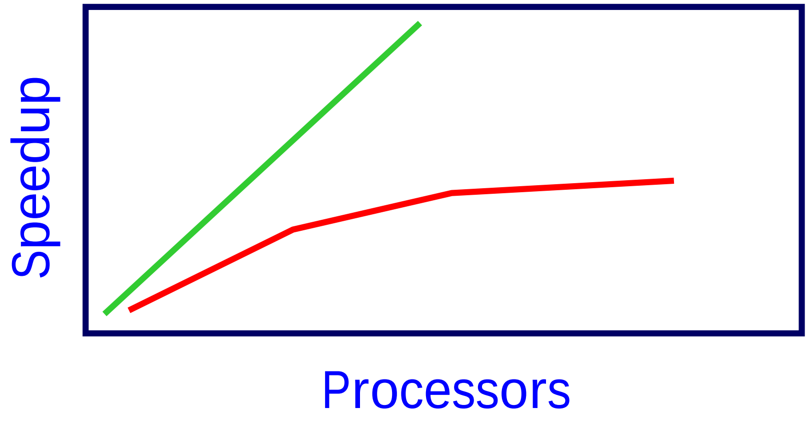
- Nudged elastic band calculations (chemistry and materials science)

- Sparse matrix-vector operations (NAS CG benchmark)

Global Array Processor Groups

If the individual calculations are small enough then each processor can be used to execute one of the tasks (embarrassingly parallel algorithms).

If the individual tasks are large enough that they must be distributed amongst several processors then the only option (usually) is to run each task sequentially on multiple processors. This usually limits the total number of processors that can be applied to the problem since parallel efficiency degrades as the number of processors increases.



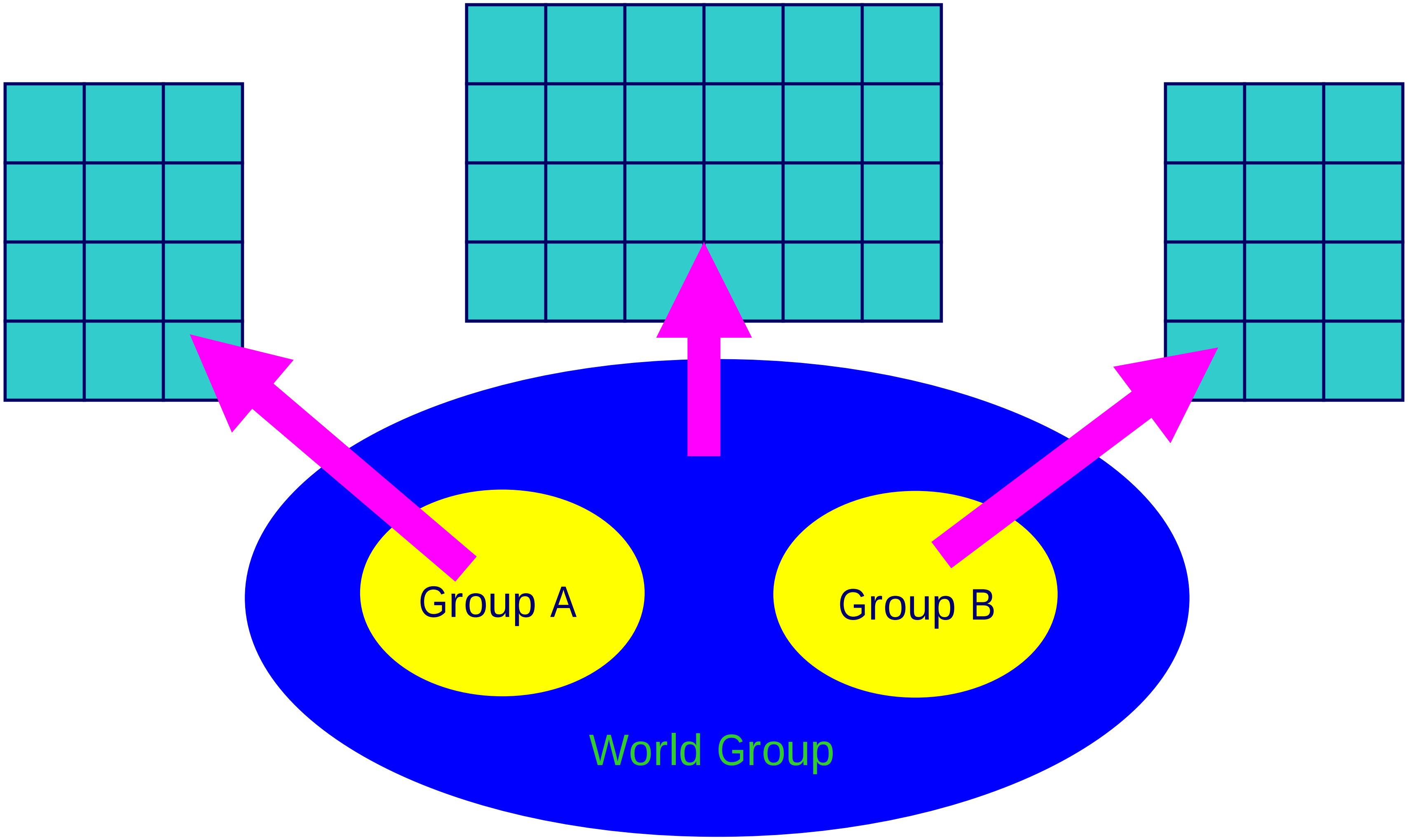
Global Array Processor Groups

Alternatively the collection of processors can be decomposed into processor groups. These processor groups can be used to execute parallel algorithms *independently* of one another. This requires

- global operations that are restricted in scope to a particular group instead of over the entire domain of processors (world group)

- distributed data structures that are restricted to a particular group

Processor Groups (Schematic)



Creating Processor Groups

integer function ga_pgroup_create(list, count)

Returns a handle to a group of processors. The total number of processors is count, the individual processor IDs are located in the array list.

subroutine ga_pgroup_set_default(p_grp)

Set the default processor to p_grp. All arrays created after this point are created on the default processor group, all global operations are restricted to the default processor group unless explicit directives are used. Initial value of the default processor group is the world group.

Explicit Operations on Groups

Explicit Global Operations on Groups

```
ga_pgroup_sync(p_grp)
ga_pgroup_brdcst(p_grp,type,buf,lenbuf,root)
ga_pgroup_igop(p_grp,type,buf,lenbuf,op)
ga_pgroup_dgop(p_grp,type,buf,lenbuf,op)
```

Query Operations on Groups

```
ga_pgroup_nnodes(p_grp)
ga_pgroup_nodeid(p_grp)
```

Access Functions

```
integer function ga_pgroup_get_default()
integer function ga_pgroup_get_world()
```

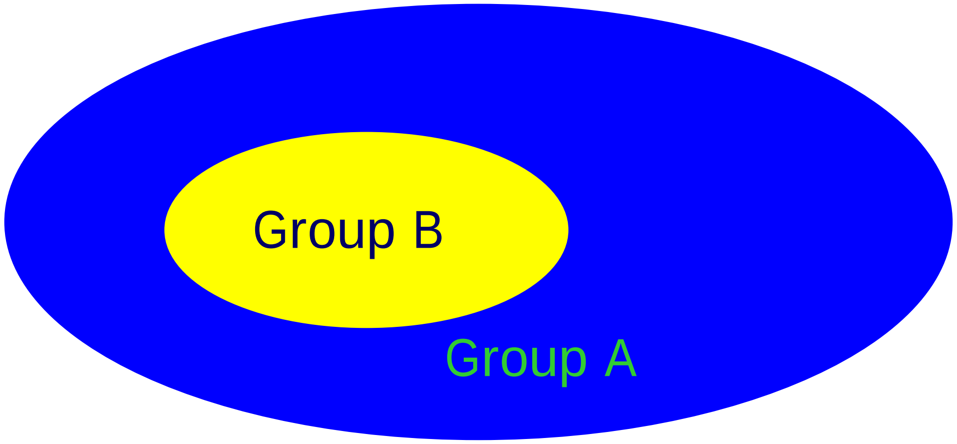
Programming with Groups

- Most explicit group operations in GA reflect operations available for MPI groups
- Concept of default group is not available in MPI
- Higher level abstractions not available in MPI

Communication between Groups

Copy and copy_patch operations are supported for global arrays that are created on different groups. One of the groups must be completely contained in the other (nested).

The copy or copy_patch operation must be executed by all processors on the nested group (group B in illustration)



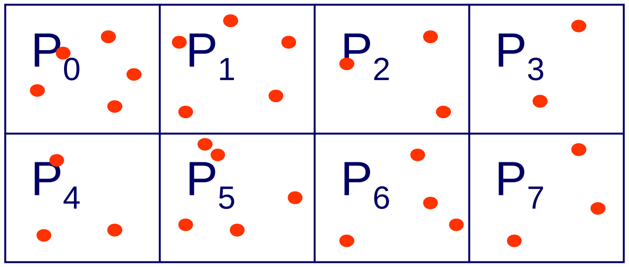
Using Processor Groups

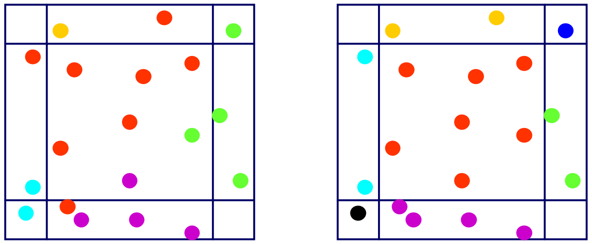
```
c     set up groups
      me = ga_nodeid()
      nprocs = ga_nnodes()
      grpsize = 4
      ngrps = nprocs/grpsize
      nproc = grpsize
      do i = 1, ngrps     ! All processors participate in
        do j = 1, grpsize ! creation of group
          proclist(j) = grpsize*(i-1) + (j-1)
        end do
        procgroup(i) = ga_pgroup_create(proclist,nproc)
      end do
      my_pgrp = (me - mod(me,grpsize))/grpsize + 1

c     run task on groups
      call ga_pgroup_set_default(procgroup(my_pgrp))
      call do_parallel_task
      call ga_pgroup_set_default(ga_pgroup_get_world())
```

MD Example

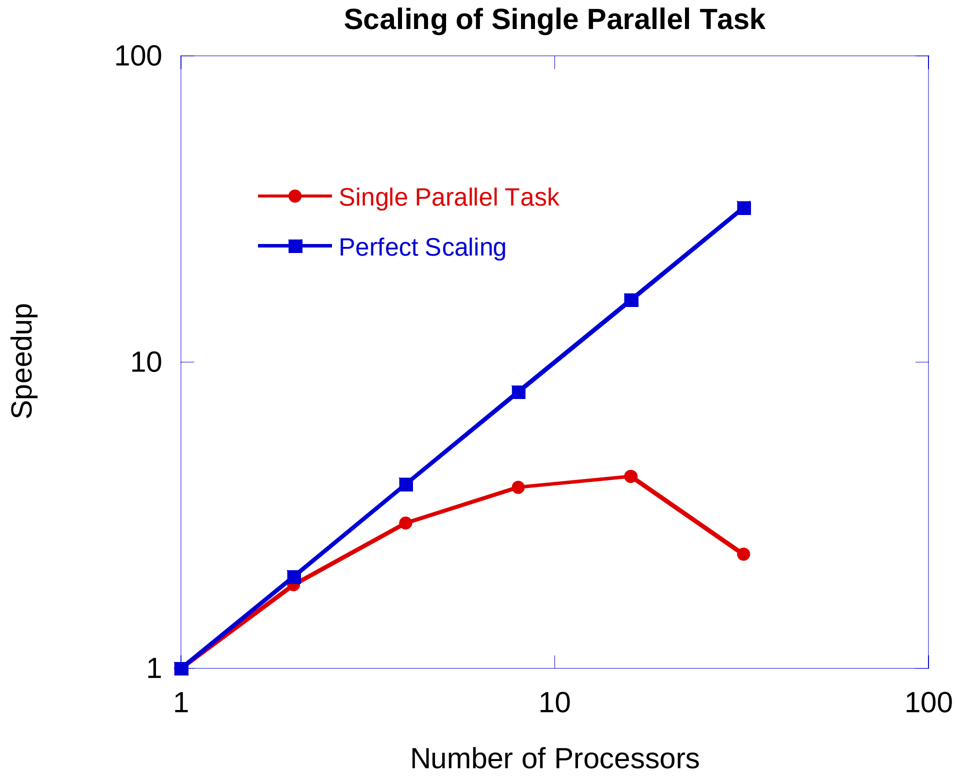
Spatial Decomposition Algorithm:

- Partition particles among processors
- Update coordinates at every step
- Update partitioning after fixed number of steps

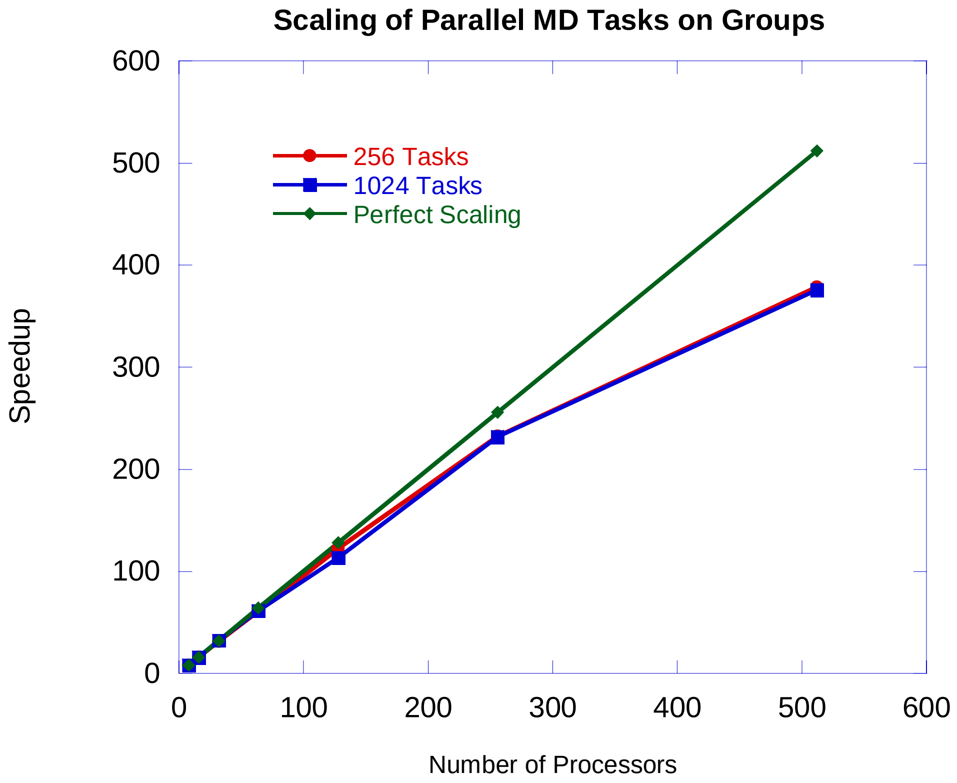




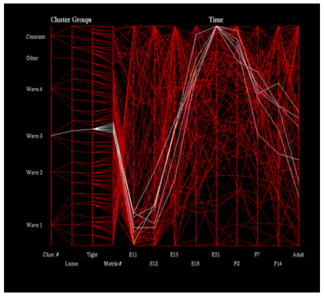
MD Parallel Scaling



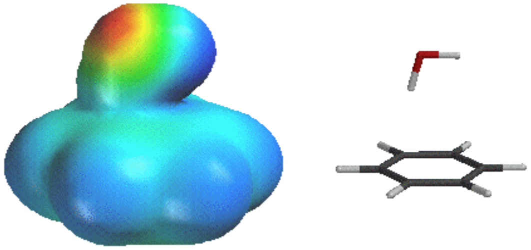
MD Performance on Groups



Application Areas

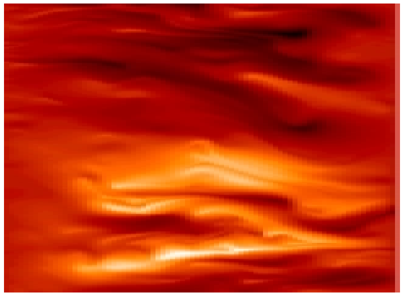


bioinformatics

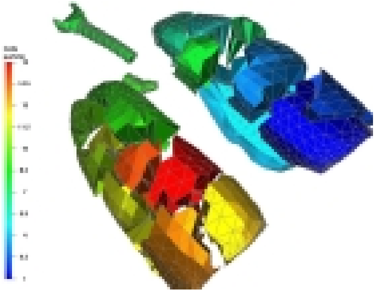


electronic structure chemistry

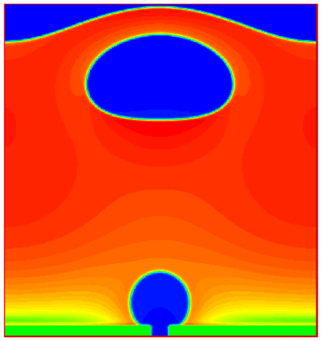
GA is the standard programming model



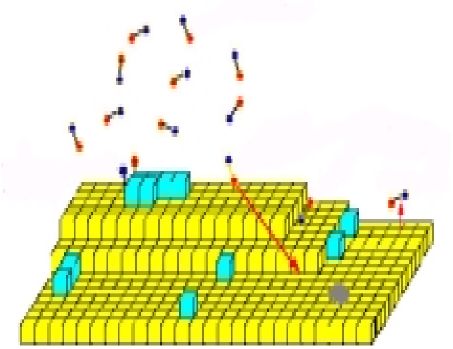
glass flow
simulation



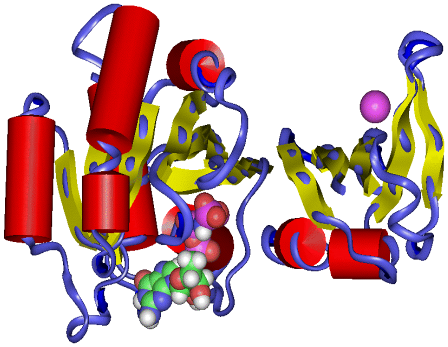
biology



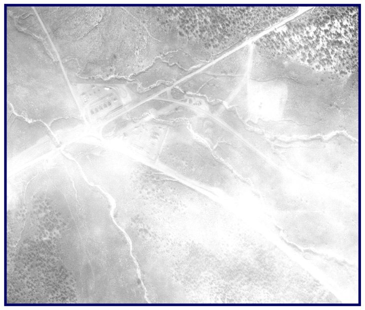
thermal flow simulation



material sciences



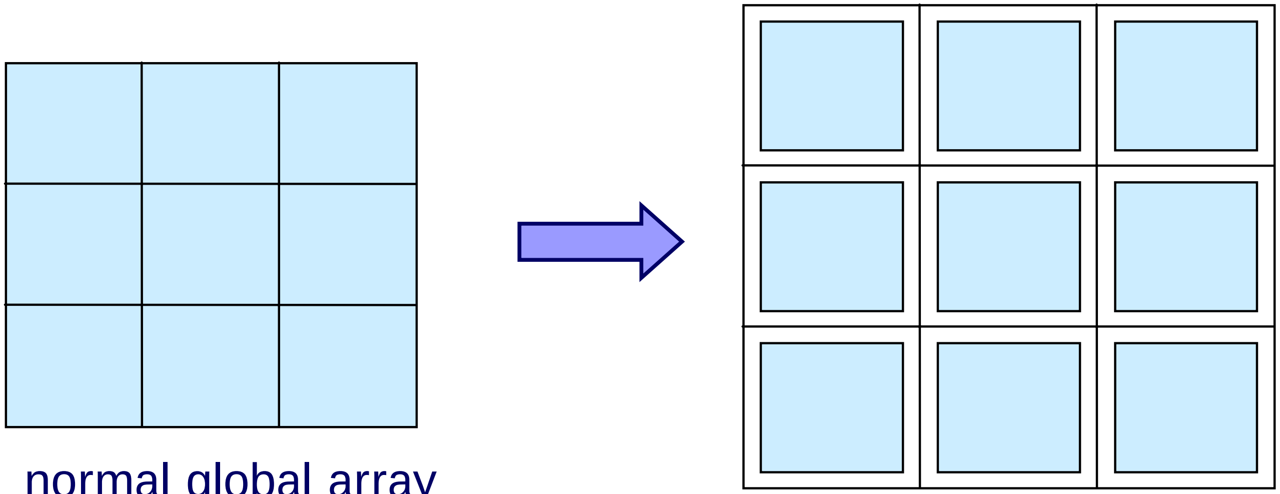
molecular dynamics



Visualization and
image analysis

Others: financial security forecasting, astrophysics, geosciences, atmospheric chemistry

Ghost Cells



normal global array

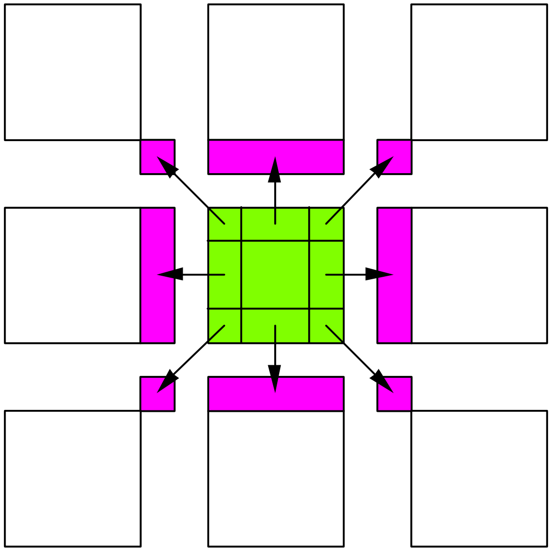
global array with ghost cells

Operations:

NGA_Create_ghosts	- creates array with ghosts cells
GA_Update_ghosts	- updates with data from adjacent processors
NGA_Access_ghosts	- provides access to “local” ghost cell elements
NGA_Nbget_ghost_dir	- nonblocking call to update ghosts cells

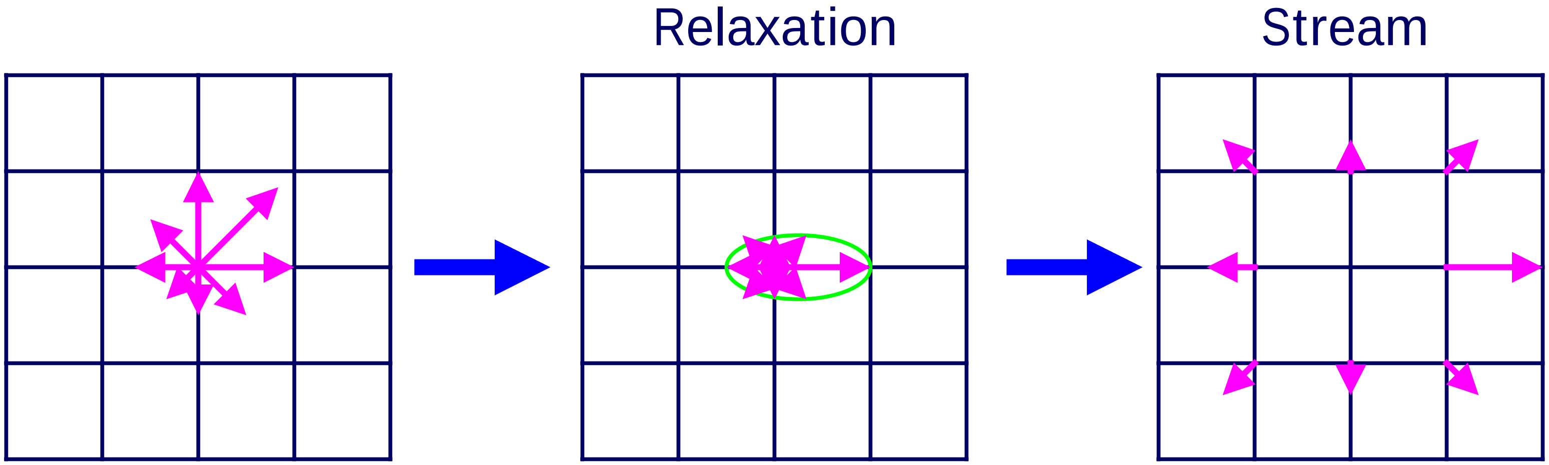
Ghost Cell Update

Automatically update ghost cells with appropriate data from neighboring processors. A multiprotocol implementation has been used to optimize the update operation to match platform characteristics.

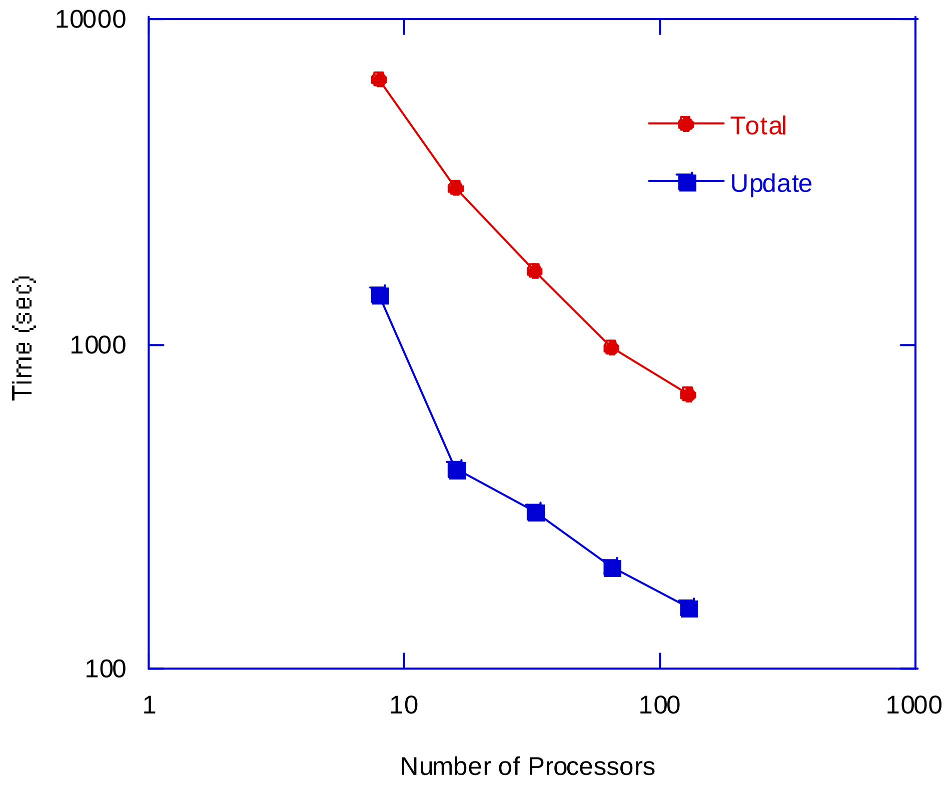


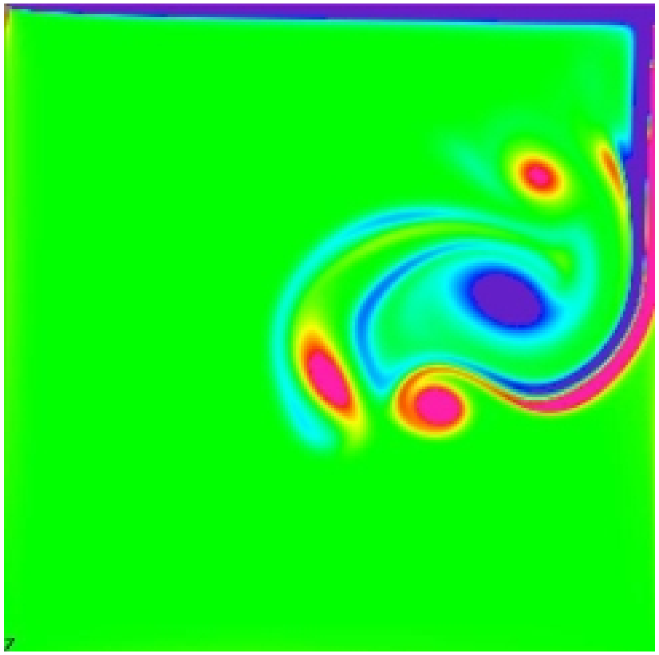
Lattice Boltzmann Simulation

$$f_i(\mathbf{r}+\mathbf{e}_i, t+\Delta t) = f_i(\mathbf{r},t) - \frac{1}{\tau}(f_i(\mathbf{r},t) - f_i^{eq}(\mathbf{r},t))$$



Ghost Cell Application Performance

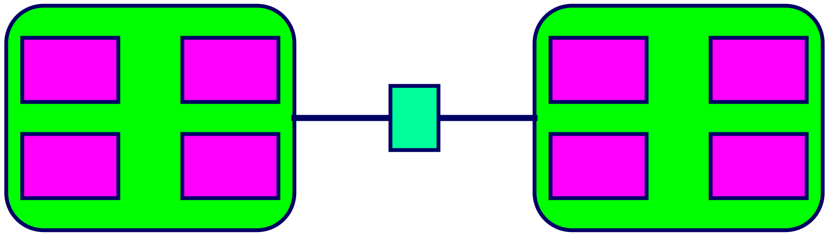


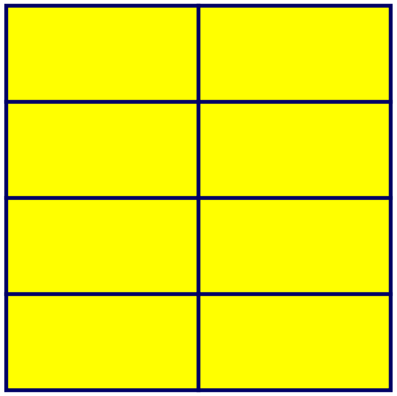


Mirrored Arrays

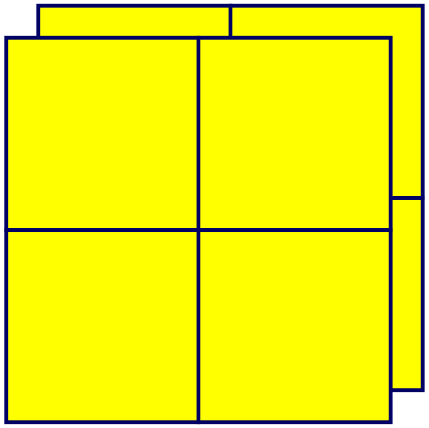
- Create Global Arrays that are replicated between SMP nodes but distributed within SMP nodes
- Aimed at fast nodes connected by relatively slow networks (e.g. Beowulf clusters)
- Use memory to hide latency
- Most of the operations supported on ordinary Global Arrays are also supported for mirrored arrays
- Global Array toolkit augmented by a merge operation that adds all copies of mirrored arrays together
- Easy conversion between mirrored and distributed arrays

Mirrored Arrays (cont.)

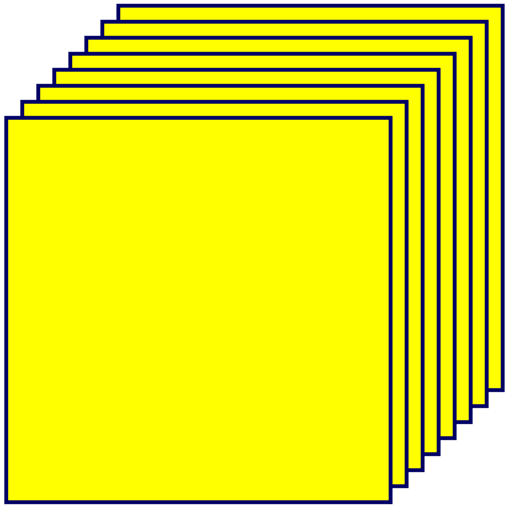




Distributed

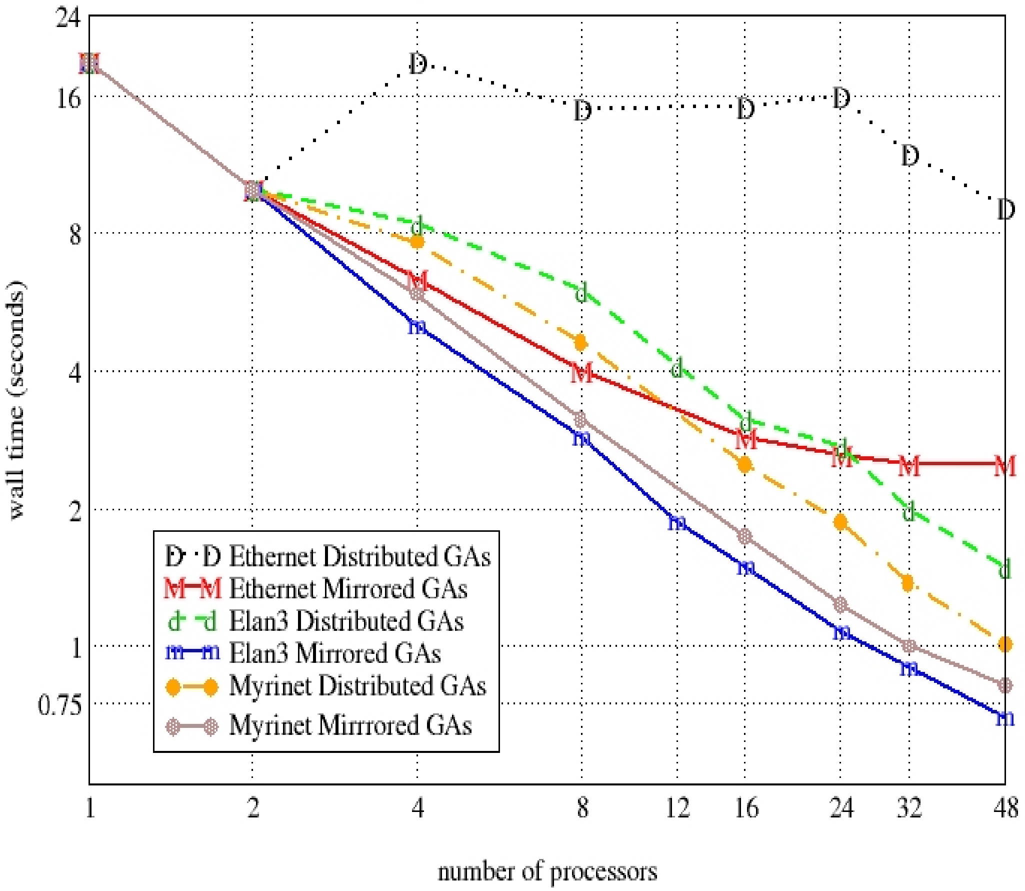


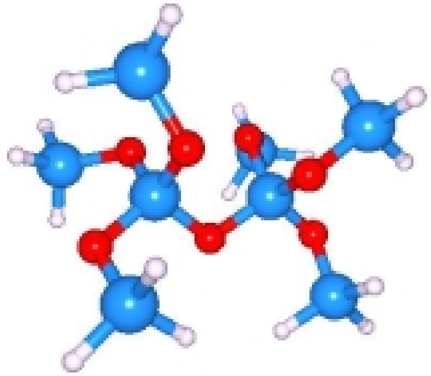
Mirrored



Replicated

NWChem DFT Calculation





http://www.emsl.pnl.gov/docs/nwchem

Other Functionality

- Common Component Architecture
- Disk Resident Arrays
  - Provide an interface between GA and distributed files on disk
- Sparse data manipulation

Related Programming Tools

- Co-Array Fortran
  - Distributed Arrays
  - One-Sided Communication
  - No Global View of Data
- UPC
  - Model Similar to GA but only applicable to C programs
  - Global Shared Pointers could be used to implement GA functionality
    - C does not really support multi-dimensional arrays
- High level functionality in GA is missing from these systems

Summary

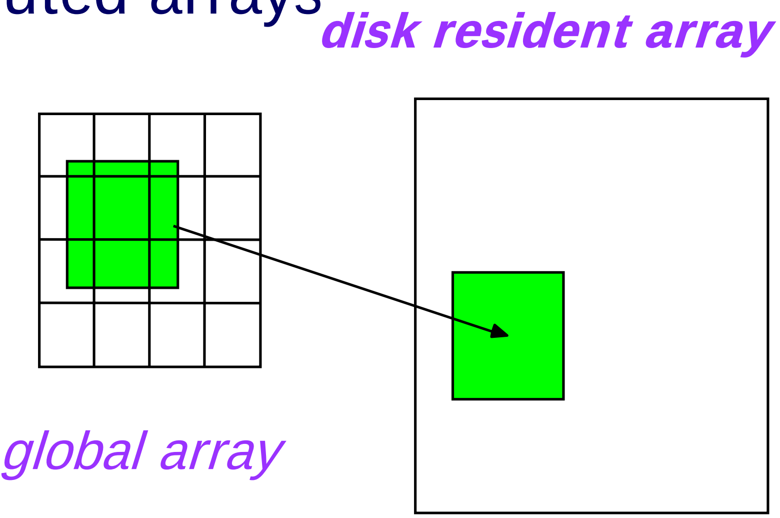
- The idea has proven very successful
  - efficient on a wide range of architectures
    - core operations tuned for high performance
  - library substantially extended but all original (1994) APIs preserved
  - increasing number of application areas
- Supported and portable tool that works in real applications
- Future work
  - Fault tolerance

Source Code and More Information

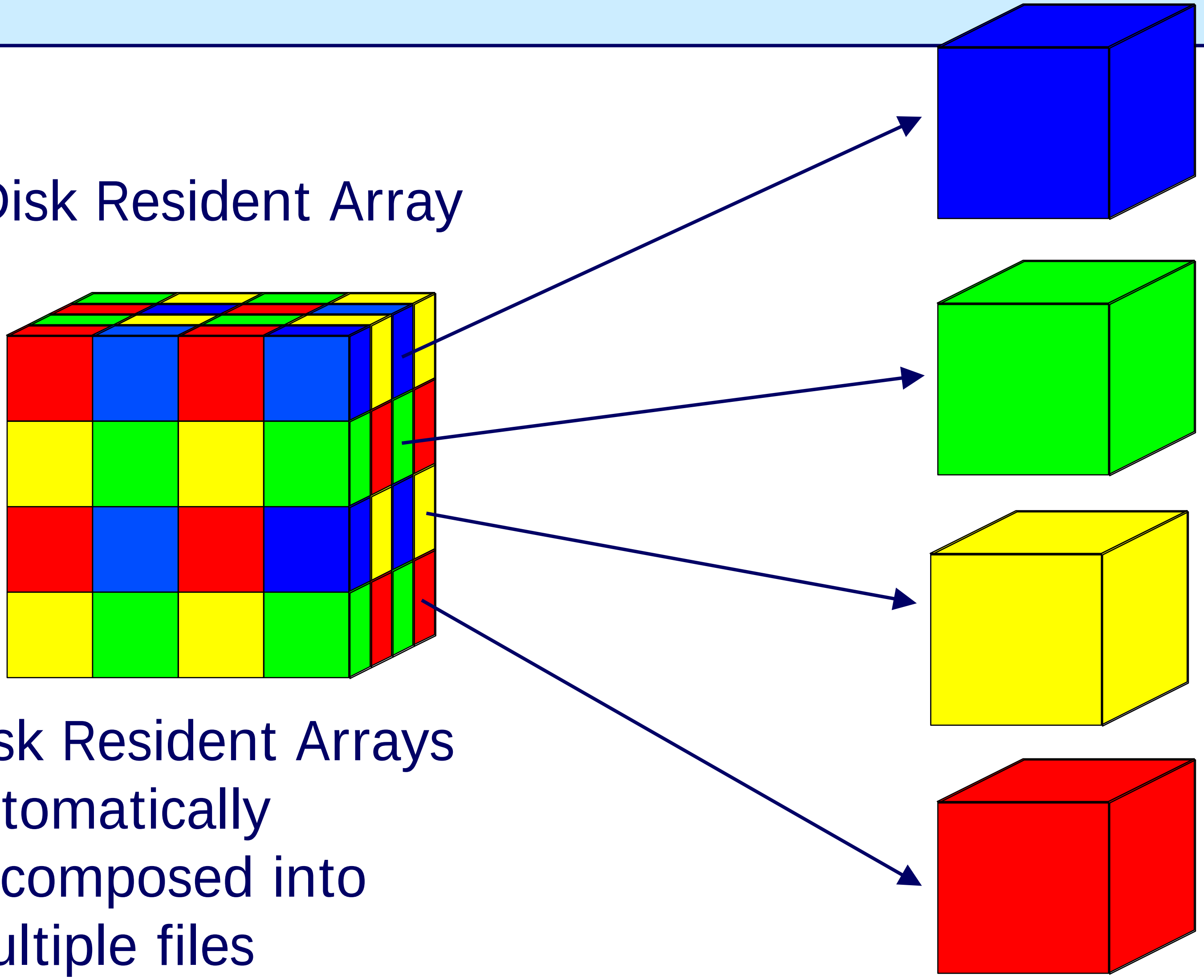
- Version 3.3 available
- Version 3.4 (with groups) available in beta
- Homepage at http://www.emsl.pnl.gov/docs/global/
- Platforms (32 and 64 bit)
  - IBM SP
  - Cray X1, XD1
  - Linux Cluster with Ethernet, Myrinet, Infiniband, or Quadrics
  - Solaris
  - Fujitsu
  - Hitachi
  - NEC
  - HP
  - Windows

Disk Resident Arrays

- Extend GA model to disk
  - system similar to Panda (U. Illinois) but higher level APIs
- Provide easy transfer of data between N-dim arrays stored on disk and distributed arrays stored in memory
- Use when
  - Arrays too big to store in core
  - checkpoint/restart
  - out-of-core solvers



High Bandwidth Read/Write

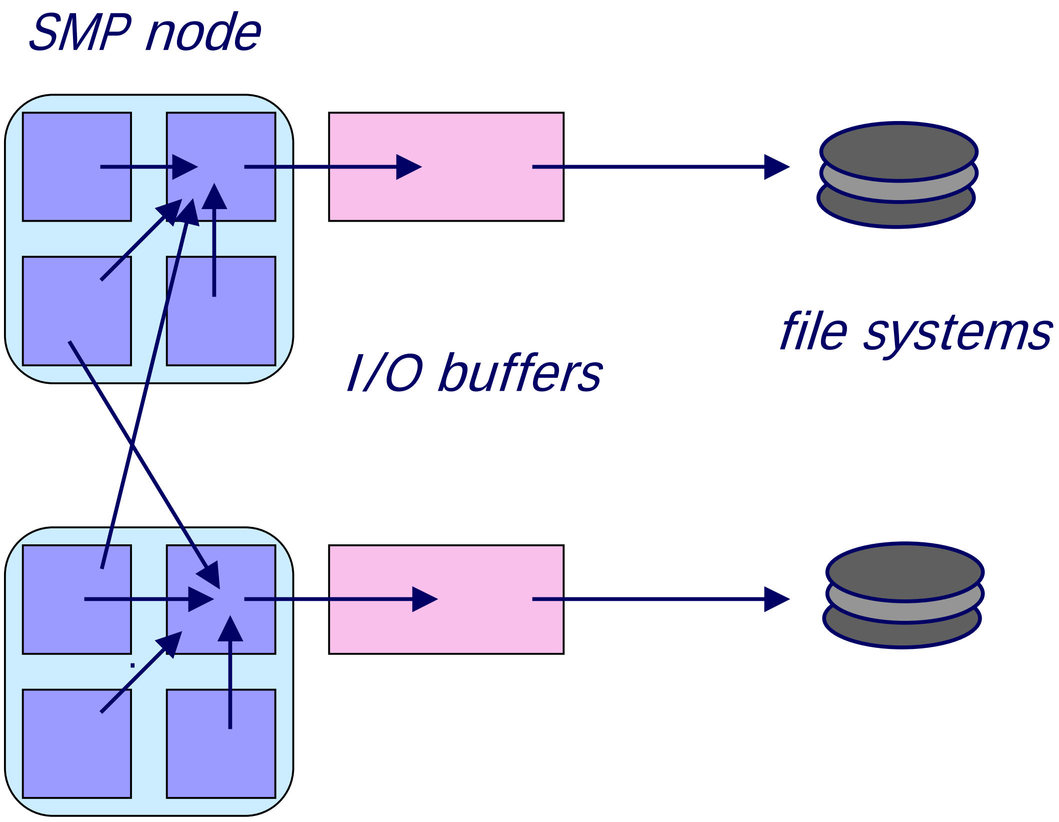


Disk Resident Array

Disks

Disk Resident Arrays automatically decomposed into multiple files

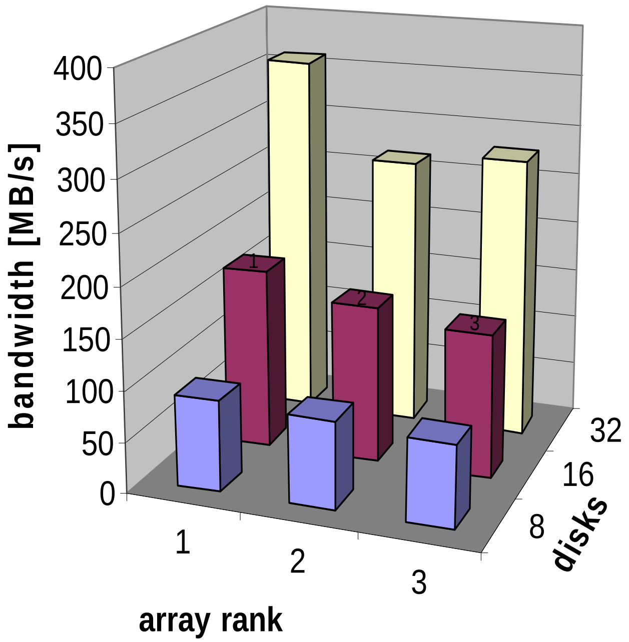
Scalable Performance of DRA



SMP node

I/O buffers

file systems



Useful GA Functions (Fortran)

```
subroutine ga_initialize()
subroutine ga_terminate()

integer function ga_nnodes()
integer function ga_nodeid()

logical function nga_create(type,dim,dims,name,chunk,g_a)
   integer type (MT_F_INT, MT_F_DBL, etc.)
   integer dim
   integer dims(dim)
   character*(*) name
   integer chunk(dim)
   integer g_a
logical function ga_duplicate(g_a,g_b,name)
   integer g_a
   integer g_b
   character*(*) name
logical function ga_destroy(g_a)
   integer g_a

subroutine ga_sync()
```

Use GA Functions (Fortran)

```
subroutine nga_distribution(g_a, node_id, lo, hi)
   integer g_a
   integer node_id
   integer lo(dim)
   integer hi(dim)
subroutine nga_put(g_a, lo, hi, buf, ld)
   integer g_a
   integer lo(dim)
   integer hi(dim)
   fortran array buf
   integer ld(dim-1)
subroutine nga_get(g_a, lo, hi, buf, ld)
   integer g_a
   integer lo(dim)
   integer hi(dim)
   fortran array buf
   integer ld(dim-1)
```

Useful GA Functions (C)

```
void GA_Initialize()
void GA_Terminate()

int GA_Nnodes()
int GA_Nodeid()

int NGA_Create(type,dim,dims,name,chunk)
   int type (C_INT, C_DBL, etc.)
   int dim
   int dims[dim]
   char* name
   int chunk[dim]
   Returns GA handle g_a
int GA_Duplicate(g_a,name)
   int g_a
   Returns GA handle g_b
   char* name
void GA_Destroy(g_a)
   int g_a

void GA_Sync()
```

Useful GA Functions (C)

```
void NGA_Distribution(g_a, node_id, lo, hi)
   int g_a
   int node_id
   int lo[dim]
   int hi[dim]
void NGA_Put(g_a, lo, hi, buf, ld)
   int g_a
   int lo[dim]
   int hi[dim]
   void* buf
   int ld[dim-1]
void NGA_Get(g_a, lo, hi, buf, ld)
   int g_a
   int lo[dim]
   int hi[dim]
   void* buf
   int ld[dim-1]
```